Self-Stabilizing Algorithms in DAG Structured Networks
Sajal K. Das, Ajoy Datta, Sébastien Tixeuil

To cite this version:

HAL Id: hal-00930899
https://hal.sorbonne-universite.fr/hal-00930899
Submitted on 14 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Self-Stabilizing Algorithms in DAG Structured Networks

Sajal K. Das\textsuperscript{1}\textsuperscript{*} Ajoy K. Datta\textsuperscript{2}\textsuperscript{†} Sébastien Tixeuil\textsuperscript{3}

\textsuperscript{1} Department of Computer Science, University of North Texas, Denton, Texas 76203
\textsuperscript{2} Department of Computer Science, University of Nevada Las Vegas, NV 89154
\textsuperscript{3} Laboratoire de Recherche en Informatique, Université de Paris Sud, France

The final version of this paper appears in [3].

Abstract

This paper describes a parameterized protocol applicable to directed acyclic graph (DAG) topologies. The function parameter of the protocol is instantiated twice to design two specific protocols: (i) the topological sorting of the successor list at every node, and (ii) a shortest path routing table construction. Both protocols are self-stabilizing and thus they are resilient to transient failures and guarantee system recovery in a finite time linear in the network diameter. From the fact that a DAG topology can be imposed on a more general topology through graph labeling protocols, the solutions presented in this paper are expected to be quite useful for a large class of distributed systems, where an optimal routing along with the robustness and fault tolerance are key factors.

1 Introduction

Robustness is one of the most important requirements of modern distributed systems since various types of (transient) faults are likely to occur as these systems are exposed to constant change of their environment.

The network topology of a distributed system can often be represented by a directed acyclic graph (DAG). In those cases where it is not a DAG, some labeling protocol may be used to transform the general topology to a DAG. Ghosh and Karaata [6] showed such a transformation protocol for networks represented by planar graphs. In systems having the DAG topology, there exist efficient solutions to many real-life problems [2], and therefore, adding robustness to these algorithms is highly desirable. One of the most inclusive approaches to fault-tolerance in distributed systems is self-stabilization [4, 7]. Introduced by Dijkstra [4], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior or the set of legitimate states. Since most self-stabilizing fault-tolerant protocols are nonterminating, if the distributed system is subject to transient faults corrupting the internal node state but not its behavior, once faults cease, the protocols themselves guarantee to recover in a finite time to a safe state without the need of human intervention. This also means that the complicated task of initializing distributed systems is no longer needed, since self-stabilizing protocols regain correct behavior regardless of the

\textsuperscript{*}Partially supported by Texas Advanced Technology Program TATP-003594031 and Texas Advanced Research Program TARPAU-97-003594-013.

\textsuperscript{†}Supported in part by a sabbatical leave grant from UNLV.
initial state. Furthermore, in practice, the context in which we may apply self-stabilizing algorithms is fairly broad since the program code can be stored in a stable storage at each node so that it is always possible to reload the program after faults cease or after every fault detection.

In this paper, we present a parameterized, self-stabilizing protocol that computes an arbitrary function $F$ in a DAG structured network. The stabilization time (and memory space used) is linear in the diameter of the network, assuming that $F$ uses constant memory variables and the maintenance of the DAG structure does not require any extra memory. Two instances of $F$ are provided as applications. The first protocol creates a topologically sorted list of reachable sites at each node, while the second one creates a shortest path routing table.

In [1], Arora et al. used the formalism of Iteration Systems to give sufficient conditions for convergence of systems, but assumed composite atomicity and fair computations. We investigate similar problems using finer grained read/write atomicity and unfair computations, but we restrict ourselves to DAG structured networks.

Our solution is also related to a recent work presented in [5] which provides a general condition for ensuring that a distributed algorithm behaves in a self-stabilizing manner in any directed graph. Therefore, this condition can be used to prove that our shortest path routing table application is indeed self-stabilizing, but still it is too restrictive to handle the case of the topologically sorted list of reachable sites application. In this paper, we investigate solutions to a broader range of problems than in [5], while making additional assumptions on the network topology.

The rest of the paper is organized as follows. Section 2 defines notations and terminology pertinent to distributed systems and self-stabilizing protocols. The topological sort and shortest path problems are also defined in this section. Section 3.1 presents our self-stabilizing, parameterized protocol and describes the functions used for the topological sort and shortest path routing protocols. The correctness reasoning and complexity issues are given in Section 4, while Section 5 offers concluding remarks.

2 Preliminaries

2.1 Model

**Distributed System.** A distributed system is modeled as a collection of processors linked with communication media allowing them to exchange information. A distributed system is a graph $S = (V, E)$, where $V$ is the set of nodes ($|V| = n$) and $E$ is the set of links or directed edges. Each node $i \in V$ represents a process$^1$, $P_i$. Each edge, denoted by an ordered pair $(i, j) \in E$, represents a communication link from $P_i$ to $P_j$, where $P_i$ is called the immediate predecessor of $P_j$ while $P_j$ is called the immediate successor of $P_i$. Nodes with no immediate predecessors (or successors) are called root (or leaf) nodes. A directed path $p_{v_0,v_k}$ between vertices $v_0$ and $v_k$ is an ordered tuple of successive links $((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$ if $k \geq 1$, or a null path $p_{v_0,v_0}$ if $k = 0$. If each $v_i$ is unique in the path, the path is elementary. A cycle is a directed path where $v_0 = v_k$. A directed graph without any cycle is a directed acyclic graph (DAG). The set of all paths between $P_i$ and $P_j$ is denoted by $X_{i,j}$. If two nodes $P_i$ and $P_j$ (not necessarily distinct) are connected by a path, $P_i$ is called a predecessor of $P_j$, and likewise, $P_j$ is a successor of $P_i$. All successors of $P_i$ (including itself) are said to be reachable.

---

$^1$For the sake of simplicity, we will use $i$ and $P_i$ interchangeably to represent the process.
from $P_i$.

**Communications.** The processors asynchronously execute their programs consisting of a set of variables and a finite set of actions. They also maintain two types of variables: *local variables* and *field variables*. The local variables cannot be accessed by any of its successors, whereas the field variables are part of the shared register which is used to communicate to the immediate successors. A processor can write only into its own shared register and can read only from the shared registers, owned by its immediate predecessors or itself. So, the field variables of a process can be accessed by the process and its immediate successors. In such a model, every processor $P_i$ is associated with two disjoint sets of registers: (i) $WR_i = \{r_{ik} | (i, k) \in E\}$ is the set of registers in which $P_i$ can write information using the `write` primitive, (ii) $RR_i = \{r_{ki} | (i, k) \in E\}$ is the set of registers in which $P_i$ can read information using the `read` primitive. Since a register $r_{ij}$ serves as a communication channel from $P_i$ to $P_j$, in the sequel, $WR_i$ will denote $P_i$'s registers. Processor $P_i$ performing `read($r_{ji}$)` atomically reads $r_{ji} \in \{RR_i \cap WR_j\}$; and performing `write($r_{ij},\text{list-of-values}$)` atomically writes list-of-values to the corresponding fields of $r_{ij} \in \{WR_i \cap RR_j\}$. Our algorithm supports read-write atomicty, thus allowing at most one `read` or `write` action in an atomic step.

**Remark 1** As such, our algorithm can be used either in true DAG systems, or in undirected systems with acyclic orientation. In a general undirected network where the node identifiers are unique, a DAG orientation can be constructed as follows. Define a total order, $\prec$, on the identifiers (id) as follows. A node $P_i$ compares its id$_i$ with that of its neighbor, say $P_j$. If id$_i$ $\prec$ id$_j$, then $P_i$ assumes that $P_j$ is one of its immediate predecessors. Otherwise, $P_j$ is one of its immediate successors. Since $\prec$ is a total order, the relations id$_i$ $\prec$ id$_j$ $\prec$ id$_k$ and id$_k$ $\prec$ id$_i$ cannot exist among any three identifiers. Therefore no cycle is possible in the resulting directed graph.

**State and Configurations.** The state of a processor is defined by the values of its local variables. The state of a link $(i, j) \in E$ is defined by the values of $r_{ij}$. A *configuration* of a distributed system $S = (V, E)$ is an instance of the states of its processors and links. The set of configurations of $S$ is denoted as $C$.

**Actions and Computations.** A processor action consists of an internal computation followed by either a `read` or a `write` action. Processor actions change the global system configuration. A *computation* $e$ of a protocol $\mathcal{P}$ is defined as a *maximal* sequence of configurations $c_1, c_2, \ldots$ such that for $i = 1, 2, \ldots$, the configuration $c_{i+1}$ is reached from $c_i$ by a single step of one process. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final global state. Note that computations considered in this paper are not assumed to be fair; only global overall progress is needed.

Let $\mathcal{C}$ be the set of possible configurations and $\mathcal{E}$ be the set of all possible computations of a protocol $\mathcal{P}$ in a distributed system $S$. Then the set of computations of $\mathcal{P}$ starting with a particular *initial configuration* $I \in \mathcal{C}$ will be denoted by $\mathcal{E}_I$. Every computation $e \in \mathcal{E}_I$ is of the form $c_1, c_2, \ldots$, with $I = c_1$. The set of computations of $\mathcal{P}$ whose initial configurations are all elements of $B \subseteq \mathcal{C}$ is denoted as $\mathcal{E}_B$. Thus, $\mathcal{E} = \mathcal{E}_C$.

**Self-stabilization.** We introduce the concept of an attractor to define self-stabilization.
Definition 1 (Attractor) Let $B_1$ and $B_2$ be subsets of $C$. Then $B_1$ is an attractor for $B_2$ if and only if $\forall b \in B_2$, $\forall e \in E_b$, $(e = c_1, c_2, \ldots)$, $\exists i \geq 1$: $c_i \in B_1$.

A specification is a predicate on computations that are admissible for a distributed system. A system matches its specification if all of its possible computations match the specification. If we consider only static problems (whose solutions are based on some global calculus), the specification can be given in terms of a set of configurations. Every computation matching the specification would be a sequence of such configurations. The topological sort or shortest path maintenance are static problems. The set of configurations that matches the specification of static problems is called the set of legitimate configurations, denoted as $L$. The remainder $C \setminus L$ denotes the set of illegitimate configurations. In the usual (i.e., non-stabilizing) distributed systems, possible computations are restricted by allowing the system to start only from some well-defined initial configurations. On the other hand, in stabilizing systems, this approach is not feasible since all possible system configurations are admissible initial configurations.

Definition 2 (Self-stabilization) A distributed system $S$ is called self-stabilizing if and only if there exists a non-empty set $L \subseteq C$ such that the following conditions hold: (i) $\forall I \in L$, $\forall e \in E_I$, $(e = c_1, c_2, \ldots)$, $\forall i \geq 1$, $c_i \in L$ (closure). (ii) $L$ is an attractor for $C$ (convergence).

Complexity. The efficiency of a self-stabilizing protocol can be measured in terms of time and memory space needed to achieve the stabilization. Since we consider asynchronous systems, in order to find the time complexity of the protocol, we must assume some bound on the communication delay between any two neighbors. We refer to this bound as a time unit. Any finite sequence of internal actions is assumed to be instantaneous. The time complexity of a self-stabilizing protocol is the time needed to reach a configuration in $L$ after the faults cease to occur. The space complexity of a self-stabilizing protocol is the memory space needed to hold the local and field variables (to maintain the information to communicate among the processors). The space complexity can be given by the number of states for the automata used to describe the system, or the number of bits required to hold the variables of the algorithm. We will use the later metric in this paper.

Programming Notation. The program for a protocol consists of a sequence of actions: $< \text{action} > \cdots < \text{action} >$. Each action has the form: $< \text{guard} > \rightarrow < \text{statement} >$. A guard is a boolean expression over the variables of a node and the field variables of its immediate successors. A statement is allowed to update the variables of the node only. Any action whose guard is true is said to be enabled. A node with one or more enabled actions is said to be privileged and may make a move executing the statement corresponding to the chosen enabled guard. We do not make any assumption on the policy adopted by a node when more than one program guards are satisfied.

Next we introduce two problems solved in this paper.

2.2 Topological Sort

The topological sort problem uses the notion of level as defined below.

Definition 3 Denoting $S_i$ as the set of immediate successors of a process $P_i$, and Max as a function that returns the maximum value of a set, the level $l_i$ of $P_i$ is defined as
\[
    l_i = \begin{cases} 
    0 & \text{if } |S_i| = 0 \\
    1 + \max_{j \in S_i} \{l_j\} & \text{otherwise.}
\end{cases}
\]

The leaf nodes (without successors) are at level 0, while a non-leaf node is at a level one plus the highest level among all of its immediate successors. For example, a node \( P_i \) with only leaves as immediate successors, will be at level \( l_i = 1 \), and a node \( P_k \) with immediate successors at levels 1, 5, and 8 will be at level \( l_k = 9 \). Every node \( P_i \) in the system computes and maintains an ordered list, \( \Gamma_i \), of its successors. That is, \( \Gamma_i = \{P_j | P_j \text{ is a successor of } P_i\} \).

We define \( L_T \) as the set of legitimate states with respect to the distributed topological sort problem, such that in any global state \( \in L_T \), the following two conditions hold:

- \( TS_1: \forall i \in V, P_j \in \Gamma_i \iff P_j \text{ is a successor of } P_i \)
- \( TS_2: \forall i \in V, \forall P_j \in \Gamma_i, P_j \text{ appears first in } \Gamma_i \text{ before all of its successors.} \)

Condition \( TS_1 \) says that \( \Gamma_i \) contains all the successors of \( P_i \) and only them. This is important since for a transient fault that corrupt memory, \( \Gamma_i \) computed by node \( i \) may not otherwise contain its correct successors. Condition \( TS_2 \) states that all reachable nodes are ordered such that predecessors of any node appear before that node in \( \Gamma_i \). Note that condition \( TS_2 \) defines only a partial order\(^2 \) on the set \( \Gamma_i \). So, there may be more than one legal states, i.e., \( |L_T| \geq 1 \).

### 2.3 Shortest Path

Suppose each edge or link \((i, j)\) has a cost, \( \text{weight}(i, j) \), associated with it. This cost may represent such link properties as the transmission time of data over the link, the link traffic, or a combination of them. The weight of a path \( p_{i,j} \) is the sum of weights of the links on this path and is denoted by \( pweight(p_{i,j}) \). In a DAG, \( pweight(p_{i,i}) = 0 \). Each node \( P_i \) in the system computes and maintains a routing table with an entry for each of its successors \( P_j \) reachable from \( P_i \). This entry contains an immediate successor of \( P_i \) and is denoted as \( \text{Path}_i^j \). If the information in all the routing tables is correct, the concatenation of the entries at all intermediate nodes for a given destination node \( P_j \) represents a directed path between \( P_i \) and \( P_j \), which is given by

\[
    \text{Path}_{i \rightarrow j} = ((i, \text{Path}_i^1 = i_1), \ldots, (i_r, \text{Path}_i^{i_r} = j)).
\]

We define \( L_S \) as the set of legitimate states with respect to the distributed shortest path routing problem, such that in any global state \( \in L_S \), the following two conditions hold:

- \( SP_1: \forall i \in V, (\text{Path}_i^j \text{ exists}) \iff (P_j \text{ is a successor of } P_i) \)
- \( SP_2: \forall \text{Path}_{i \rightarrow j}, \ pweight(\text{Path}_{i \rightarrow j}) = \min_{x_{i,j} \in X_{i,j}} \{pweight(x_{i,j})\}, \) where \( X_{i,j} \) is the set of all paths between \( P_i \) and \( P_j \).

Condition \( SP_1 \) says that the routing table contains an entry for each \( P_j \) which is a successor of \( P_i \), while Condition \( SP_2 \) states that any path constructed with the routing table is of minimal cost. Since there may be more than one minimal path between \( P_i \) and \( P_j \), we have \( |L_S| \geq 1 \). Figure 1 presents an example of a global state satisfying both \( SP_1 \) and \( SP_2 \).

\(^2\)We can extend it to a total order relation by defining an ordering among the node identifiers when their level information is equal.
3 Proposed Protocols

In this section, we first provide a parameterized self-stabilizing algorithm that computes a global function $\mathcal{F}$ on a DAG network. We then present two instances of the function, namely $\mathcal{F}_{TS}$ and $\mathcal{F}_{SP}$, which solve the topological sort and shortest path problems, respectively.

3.1 Parameterized Protocol

The parameterized protocol uses three state variables for each processor $P_i$: (a) $S_i$ is the set of immediate successors of node $P_i$; (b) $\beta_i$ is the set of two-tuples $(P_j, \gamma_j)$ where $P_j$ is a successor of $P_i$ and $\gamma_j$ is a variable of arbitrary type; and (c) $C_i$ is the array of sets containing the most recently copied $\beta_k$ for each $P_k \in S_i$. The type of the set $\beta_i$ is denoted as $\mathbb{L}$, while those of the $C_i$ array is denoted as $\mathbb{C}$. Underlined variables (such as $\underline{\beta_i}$) are field variables which can be written by a single node but read by all of its immediate predecessors. In addition to the above state variables, the protocol maintains a function parameter $\mathcal{F}$ whose domain is $\mathbb{C}$ and co-domain is $\mathbb{L}$. The function $\mathcal{F}$ is a well-defined recursive local function and it associates to each $C_i \in \mathbb{C}$ its image $\mathcal{F}(C_i) \in \mathbb{L}$. The input to $\mathcal{F}$ are the values of the state variables copied from the immediate successors and the outputs are the current state of the node computing the function. The parameterized algorithm consisting of two rules instantiated by $\mathcal{F}$ is as shown below.

**Algorithm 3.1 $\mathcal{F}$-Parameterized Algorithm on DAG**

$PP_1[\mathcal{F}]:: (P_k \in S_i ; (\alpha := \text{read}(\beta_k)) \wedge C_i[k] \neq \alpha \rightarrow C_i[k] := \alpha;$

$PP_2[\mathcal{F}]:: (\text{read}(\beta_i) \neq \mathcal{F}(C_i)) \rightarrow \text{write}(\beta_i, \mathcal{F}(C_i));$

Rule $PP_1[\mathcal{F}]$ copies the immediate successor field variables into the node’s local variables, so that they can be used at a later time. It uses a local variable $\alpha$ to avoid reading the field variable $\beta_k$ twice, and thus, implements the read/write atomicity. In $PP_1[\mathcal{F}]$, an expression of the form <left> := <right> is used in which the operand left must be a variable while right may be a constant or a variable. The operator := assigns the value of right to left.
and always returns true. Rule $PP_2[\mathcal{F}]$ computes the $\mathcal{F}$ function with the previously copied variables as input.

### 3.2 Topological Sort Protocol

Since the topological sort problem requires that $\beta_i$s are ordered by levels, we replace the variable $\gamma_j$ in the parameter function with $l_j$. In other words, we assume that the set $\beta_i$ contains tuples $\langle P_j, l_j \rangle$, where $P_j$ is a successor of $P_i$ and $l_j$ its level number.

Let us define two helper functions: $\text{Max}$ returns the maximum level among all the tuples in the $C_i$ array. Whereas $\text{Sort}$ sorts a set of tuples $\langle P_i, l_i \rangle$ in descending order of $l_i$, i.e., $\langle P_j, l_j \rangle > \langle P_k, l_k \rangle$ if and only if $l_j > l_k$. We instantiate the function $\mathcal{F}$ for the topological sort protocol with $\mathcal{F}_{TS}(C_i)$ such that:

$$
\mathcal{F} = \begin{cases} 
\{ \langle P_i, 0 \rangle \} & \text{if } |C_i| = 0 \\
\text{Sort} \left( \left\{ \langle P_i, 1 + \max\{ (P_j, l_j) \in C_i[k] \land P_k \in S_i \} \{ l_j \} \rangle \right\} \cup \bigcup_{P_k \in S_i} C_i[k] \right) & \text{otherwise}
\end{cases}
$$

where $C_i[k]$ holds the most recently copied value of $\beta_k$. In a correct configuration, the set $\beta_i$ of every $P_i$ contains one entry for each of its successor nodes, plus one entry for $P_i$ itself. Moreover, every $l_j$ in a tuple $\langle P_j, l_j \rangle$ in every $\beta_i$ is equal to the level of node $P_j$. A leaf node $P_i$ (with $|C_i| = 0$) continually updates $\beta_i$ with $\langle P_i, 0 \rangle$ according to Rule $PP_2[\mathcal{F}_{TS}]$, because a leaf node can only reach itself by definition. In addition, a leaf node cannot execute Rule $PP_1[\mathcal{F}_{TS}]$ since it does not have any immediate successor. A non-leaf node $P_i$, on detecting an inconsistency between its internal copy $C_i[k]$ and the actual value $\beta_k$ of an immediate successor node $P_k$, updates the $C_i[k]$ entry to reflect the new information by Rule $PP_1[\mathcal{F}_{TS}]$. Finally, $P_i$ recomputes $\beta_i$ based on the most recent information in every entry in $C_i$, by Rule $PP_2[\mathcal{F}_{TS}]$. In this last step, we must make sure that the $\mathcal{F}_{TS}$ function properly calculates the level of node $P_i$ and the topologically sorted list of $P_i$’s successors.

**Example.** Consider the DAG structured network consisting of seven nodes (numbered from $A$ to $G$) as shown in Figure 2. A leaf node $(E, F, G)$ only owns a single tuple $\langle P_i, 0 \rangle$ since for leaf nodes, $|C_i| = 0$. Nodes $C$ and $D$ are at level 1 since their first entry in the sorted list is $\langle P_i, l_i \rangle$, and nodes $A$ and $B$ are at level 2. Note that each node contains the list of nodes whose level is below theirs, and since the $\text{Sort}$ function operates on levels, the highest level (i.e., that of the current node) appears first.

### 3.3 Shortest Path Protocol

For this protocol, the set $\beta_i$ of Algorithm 3.1 contains tuples $\langle P_j, (d_j, b_j) \rangle$ where $P_j$ is a processor reachable from $P_i$. In other words, we replace the variable $\gamma_j$ by $(d_j, b_j)$, where $d_j$ is a real variable representing the distance to node $P_j$ from node $P_i$, and $b_j$ contains a processor id $P_k \in S_i$ such that $P_k$ is the best successor (i.e., on the path of minimal cost) to route a message from $P_i$ to $P_j$.

To implement the shortest path protocol, we define several operators. For example, $+d$ is a binary operator that applies on a list $l$ of tuples and a scalar $w$. Computing $l + d w$ makes each tuple $\langle P, (d, b) \rangle$ in $l$ become $\langle P, (d + w, b) \rangle$. The binary operator $:= b$ applies on a list...
$l$ of tuples and a processor $P_k$. Computing $l = b P_k$ makes each tuple $⟨P, (d, b)⟩$ in $l$ become $⟨P, (d, P_k)⟩$. The operator $\text{Sort}$ sorts a set of tuples $⟨P_i, (d_i, b_i)⟩$ in the ascending order of $d_i$, i.e., $⟨P_j, (d_j, b_j)⟩ < ⟨P_k, (d_k, b_k)⟩$ if and only if $d_j < d_k$.

We instantiate the function $\mathcal{F}$ for the shortest path protocol with $\mathcal{F}_{SP}(C_i)$ such that:

$$\mathcal{F} = \begin{cases} 
\{⟨P_i, (0, P_i)⟩\} & \text{if } |C_i| = 0 \\
\text{Sort}\left(\{⟨P_i, (0, P_i)⟩\} \cup \bigcup_{P_k \in S_i} \{(C_i[k] + \text{weight}(i, k)) := b P_k\}\right) & \text{otherwise}
\end{cases}$$

We now define a function $\text{Next}$ which uses $\beta_i$ to generate the routing table at node $P_i$: $\text{Next}(i, j) \equiv \text{First}(P_j, (d_j, P_k)) \in \beta_i P_k$. In the function $\overline{\text{Next}}(i, j)$, $\text{First}_{\text{condition}(A)} A$ returns the first occurrence of $A$ such that $\text{condition}(A)$ is true. Intuitively, $\text{Next}(i, j)$ returns the immediate successor $P_k$ of $P_i$ such that $P_k$ is the next node on a minimal path from $P_i$ to $P_j$. Thus, the function $\text{Next}(i, j)$ returns $\text{Path}_i^j$. Since the function $\text{Next}$ needs to access a single field variable ($\beta_i$) only once, it can be considered as being executed atomically. In a correct configuration, the set $\beta_i$ of each $P_i$ contains one entry for each of its successors, plus one entry for $P_i$ itself. According to Rule $PP2[\mathcal{F}_{TS}]$, $P_i$ computes $\beta_i$ to contain the correct value of $\beta_i$ based on the most recent information in every entry in $C_i$. In this last step, we make sure that the function $\mathcal{F}_{SP}$ properly calculates the distance to the successors of $P_i$ and the best immediate successor to reach successors of $P_i$.

**Example.** Consider the same example of a DAG network as in Figure 2. In Figure 3, a leaf node ($E$, $F$, or $G$) only owns a single tuple $⟨P_i, (0, P_i)⟩$ since for leaf nodes, $|C_i| = 0$. Note that weights may vary from one edge to the other. Node $B$ contains two entries $⟨E, ⟨3, C⟩⟩$ and $⟨E, ⟨5, D⟩⟩$ relative to node $E$, where $⟨E, ⟨3, C⟩⟩$ means that node $E$ is reachable from node $B$ with weight 3 passing through node $C$, while $⟨E, ⟨5, D⟩⟩$ means that node $E$ is reachable from node $B$ with weight 5 passing through node $D$. Since the tuples are sorted according to the weight of each tuple, the $\text{Next}$ function returns the neighboring node that is on a least cost path.
4 Correctness Reasoning

This section provides the correctness proof for the parameterized algorithm and analyzes its space and time complexity. The same reasoning as in [5] holds as far as the read/write atomicity is concerned. We also present proofs for the correctness of the functions defined in Sections 3.2 and 3.3. We assume that the input to these functions is maintained (or corrected) by an underlying stabilizing protocol.

4.1 Correctness of the Parameterized Protocol

In the following proofs, we assume that rules are executed atomically. Let us show that our algorithm also works if we assume the read/write atomicity model. Let $S_F$ denotes the system executing Algorithm 3.1 instantiated with the function $F$.

**Lemma 1** Rules $PP_1 | F$ and $PP_2 | F$ of System $S_F$ can be considered as atomic.

**Proof:** Rule $PP_1 | F$ contains only one read statement and some internal actions, and hence, can be considered as atomic in the read/write atomicity model.

Rule $PP_2 | F$ contains one read statement and one write statement, both using the same register $r$. In the read/write atomicity model, these two actions can be interleaved with other processor actions. Now we will prove that any interleaving in any computation is equivalent to a computation where rule $PP_2 | F$ is executed atomically.

Let $P_i$ be the owner of $r$ and $N(i)$ denote the set of $P_i$'s neighbors. Interleaving a Rule $PP_2 | F$ at $P_i$ and Rule $PP_2 | F$ at $P_j$ is equivalent to considering them as executing atomically since they apply to different registers ($PP_2 | F$ apply both of its two actions to the same register). For the same reason, interleaving $PP_2 | F$ at $P_i$ and $PP_1 | F$ at $P_j \notin N(i)$ is equivalent to considering them as executing atomically.

There remains the case when Rule $PP_2 | F$ at $P_i$ is interleaved with Rule $PP_1 | F$ at $P_j \in N(i)$. The two actions may involve the same register $r$, and the resulting computation would be as follows: ..., $\text{read}_1(r), \text{read}_2(r), \text{write}_3(r), ...$. Now this computation is equivalent to the following, where rule $PP_2 | F$ (that includes $\text{read}_1(r)$ and $\text{write}_3(r)$ actions) is executed
atomicity: \ldots, read_j(r), read_i(r), write_i(r)\ldots. In any case, any interleaved computation is equivalent to a computation where Rule PP_2[F] is executed atomically. 

Let \( N_k \) be the set of nodes at level \( k \). We define a set of legitimate configurations \( L_F \) of the system \( S_F \) as follows:

\[
L_F \equiv \left\{ \forall i \in N_0, \beta_i = F(C_i), \text{ with } |C_i| = 0 \right\}
\]

We prove the following two lemmas.

**Lemma 2 (Closure)** Starting from a configuration in the legitimate set \( L_F \) and using a properly defined \( F \), any computation of the system \( S_F \) remains in \( L_F \).

**Proof:** Since \( C_i[k] = \beta_k \) holds \( \forall k \in S_j \forall i \in N_j, \forall j \geq 1 \), Rule PP_2[F] is not applicable at any node. Also, \( \forall i \in N_0, \beta_i = F(C_i) \) holds. So, Rule PP_2[F] is not applicable at any leaf node. \( \forall j \geq 1, \forall i \in N_j, \beta_i = F(C_i) \) holds, implying Rule PP_2[F] is not applicable at any non-leaf node. Since no rule is applicable at any node, the configuration (in \( L_F \)) remains the same. 

**Lemma 3 (Convergence)** Starting from any initial configuration and using a properly defined \( F \), any computation of the system \( S_F \) eventually reaches a configuration in \( L_F \).

**Proof:** We define a variant function \( M \) that returns zero or a positive integer value, and show that \( M \) is strictly decreasing for the execution of any rule by any node. Furthermore, when \( M \) returns zero for any level, the system has reached a legitimate configuration. Let \( \delta \) be the maximum out-degree of any node in the DAG network. Then the function \( M \) is defined for the set of nodes in \( N_i \) as follows:

\[
M(N_i) \equiv \left\{ \sum_{j \in S_i} \phi_{j \neq F(C_i)} + \sum_{k \in S_j} 2 \cdot \phi_{C_j[k] \neq \beta_k} \right\}
\]

where \( \phi_A \) is a characteristic function for the condition \( A \) such that \( \phi_A \) returns 1 if \( A \) is true and 0, otherwise. For example, if for some \( j \in N_i \), there exists a \( k \in S_j \) such that \( C_j[k] \neq \beta_k \), then \( \phi_{C_j[k] \neq \beta_k} \) returns 1. The sum \( \sum_{k \in S_j} 2 \cdot \phi_{C_j[k] \neq \beta_k} \) can be interpreted as equal to “twice the number of \( j \)’s immediate successors \( k \) such that \( \beta_k \) does not match \( j \)’s view of \( \beta_k \).”

The following claim helps prove our lemma.

**Claim:** It holds the property \( R(k) \equiv \forall e \in E, (e = c_1, c_2, \ldots), \exists i \geq 1, \forall j \geq i, M(N_k) = 0 \) in \( c_j \).

**Proof** (by induction on the level of nodes):

**Base Case:** To prove \( R(0) \), we have to show that \( \forall e \in E, (e = c_1, c_2, \ldots), \exists i \geq 1, \forall j \geq i, M(N_0) = 0 \) in \( c_j \). The set \( N_0 \) contains the leaf nodes and have no immediate successors. So, \( \forall i \in N_0, |C_i| = 0 \) and \( M(N_0) = \sum_{j \in N_0} \phi_{\beta_i \neq F(C_i)} \). The condition in the characteristic function \( \phi \) is precisely the guard of the only rule PP_2[F] applicable to a leaf node. This rule can be applied only once at any leaf node and every such application decreases \( M(N_0) \) by 1. When no rules are enabled at any leaf node, \( M(N_0) = 0 \). Since nodes in \( N_0 \) have no immediate successors, no rules can be applied at this state. So, \( M(N_0) \) remains equal to 0.
for any subsequent computation. Thus, \( R(0) \) holds.

**Induction Step:** Assume that there exists a \( k \leq h - 1 \) such that \( \forall j \ (0 \leq j \leq k) \), \( R(j) \) holds where \( h \) is the height of the DAG network. We need to show that \( R(k + 1) \) also holds. From the induction hypothesis, starting from an arbitrary initial configuration, the system \( S_F \) eventually reaches a configuration, \( c_k \), where \( \forall j \ (0 \leq j \leq k) \), \( \mathcal{M}(N_j) = 0 \) and remains equal to 0 in any subsequent computation. We now consider the nodes in \( N_{k+1} \), whose direct successors are at level \( \leq k \). The immediate successors cannot apply any rules since \( \forall j \ (0 \leq j \leq k) \), \( \mathcal{M}(N_j) = 0 \). Consider any node \( m \in N_{k+1} \) reading the variables \( \beta \) of \( i \)'s successors from \( m \)'s immediate successors. Since no successor of \( m \) can execute any rule, the variables \( \beta \) read by \( m \) remains unchanged from the current configuration, \( c_k \), onwards. Since the nodes in the immediate successor set \( S_m \) do not change \( \beta \), the node \( m \) can execute Rule \( PP_1[\mathcal{F}] \) at most \( \delta \) times. Rule \( PP_2[\mathcal{F}] \) can be applied only once at \( m \) unless any node \( j \in S_m \) changes the array \( C_j \) due to the execution of Rule \( PP_1[\mathcal{F}] \). (Note that \( C_j \) is an input to \( \mathcal{F} \).) From the definition of \( \mathcal{M}(N_i) \), there is a direct relation between the execution of the rules in Algorithm 3.1 and the characteristic functions. Every execution of Rule \( PP_2[\mathcal{F}] \) decreases \( \mathcal{M}(N_{k+1}) \) by 1, while every execution of Rule \( PP_1[\mathcal{F}] \) decreases \( \mathcal{M}(N_{k+1}) \) by either 1 or 2 depending on if it subsequently enables Rule \( PP_2[\mathcal{F}] \) or not. So, \( \mathcal{M}(N_{k+1}) \) strictly decreases every time a rule is executed by a node in \( N_{k+1} \). Eventually, \( \mathcal{M}(N_{k+1}) \) becomes equal to 0 and both rules are disabled at every node in \( N_{k+1} \). Since successors of nodes in \( N_{k+1} \) cannot change their \( \beta \) variables, both rules remain disabled forever and \( \mathcal{M}(N_{k+1}) \) remains equal to 0 for any subsequent computation. Thus, \( R(k + 1) \) holds. Hence the claim.

The value of \( \mathcal{M}(N_i) \) being 0 for any level \( i \) of the DAG network implies that all characteristic functions used in the definition of \( \mathcal{M}(N_i) \) return 0. Thus, the system \( S_F \) reached a configuration in \( \mathcal{L}_F \). Hence the proof of Lemma 4.3.

**Theorem 1 (Stabilization)** The system \( S_F \) is self-stabilizing.

**Proof:** Follows from Lemmas 2 and 3.

### 4.2 Correctness of Functions \( \mathcal{F}_{TS} \) and \( \mathcal{F}_{SP} \)

From Theorem 1, the system \( S_{\mathcal{F}_{TS}} \) (respectively, \( S_{\mathcal{F}_{SP}} \)) is self-stabilizing, i.e., it is guaranteed to converge to a configuration in \( S_{\mathcal{F}_{TS}} \) (respectively, \( S_{\mathcal{F}_{SP}} \)), where all inputs to the function \( \mathcal{F}_{TS} \) (respectively, \( \mathcal{F}_{SP} \)) are correct. We now show that \( \mathcal{F}_{TS} \) (respectively, \( \mathcal{F}_{SP} \)) returns correct results when fed with a correct input.

**Lemma 4** The function \( \mathcal{F}_{TS} \) solves the Topological Sort problem.

**Proof:** We need to prove that the function \( \mathcal{F}_{TS} \) creates the set \( \beta \) at every node such that Conditions \( TS_1 \) and \( TS_2 \) hold. First, we show that for any node \( P_i \) in the network, the set \( \beta_i \) contains at least one tuple for every successor, \( P_j \), of \( P_i \) (including \( i \)) such that each tuple contains the identifier, \( P_j \), of the successor and the level \( \beta_j \) of \( P_j \). We prove this by induction on the level of the nodes contained in the DAG network. For a leaf node \( P_i \), \( |C_i| = 0 \) and \( \mathcal{F}_{TS} \) returns \( \langle P_i, 0 \rangle \). This is correct since the leaf nodes are always at level 0. For a non-leaf node, \( P_i \), assuming that the input from all its immediate successors is correct, the set \( \beta_i \) is added to the union of all \( \beta_j \)'s read from every \( P_j \in S_i \). Thus, \( \beta_i \) contains the tuples of all successors
of \( P_i \) except \( P_i \) itself. The function \( F_{TS} \) creates a new tuple \( (P_i, l_i) \) such that \( l_i \) is one plus the maximum of the levels of all immediate successors of \( P_i \) (by the induction hypothesis). So, by definition, \( l_i \) is equal to the level of node \( P_i \). Next, we prove that \( \beta_i \) is topologically sorted, i.e., every node, \( P_j \) appears in \( \beta_i \) before all of \( P_j \)’s successors. Since \( F_{TS} \) sorts \( \beta_i \) in descending order of node-levels, the node with the highest level is placed first in the list. Since any non-root node has a level at least one higher than than any of its immediate successors (and obviously, their successors), the set \( \beta_i \) is topologically sorted.

Lemma 5 The functions \( F_{SP} \) and \( \text{Next} \) solve the Shortest Path problem.

Proof: We need to prove that the function \( F_{SP} \) creates the set \( \beta \) at every node such that the function \( \text{Next} \) satisfies Conditions \( TS_1 \) and \( TS_2 \). The proof of correctness of Function \( F_{SP} \) is similar to that of \( F_{TS} \), and hence is omitted here. We now show that the function \( \text{Next} \) implements a shortest path routing table. Since the tuples in \( \beta_i \) are sorted in ascending order of distance, the shortest distance is ranked first in the list. From the definition of \( \text{Next} \), it returns the first tuple in \( \beta_i \) which matches the destination. Thus, the next node returned by \( \text{Next} \) will be on a shortest path to the destination. In addition, as we discussed in Section 3.3, the \( \text{Next} \) function could be executed atomically.

4.3 Complexity Results

Theorem 2 (Space Complexity) The average amount of space used per node of the system \( S_F \) is \( O(D \times s) \), where \( D \) is the diameter of the DAG network and \( s \) is the size of a \( \beta_i \) tuple.

Proof: Consider a legitimate configuration. Every leaf node \( i \) at level 0 maintains a tuple for itself in \( \beta_i \), and has at most \( n - 1 \) predecessors (for example, in a chain) that do the same. This creates at most \( n \) tuples in the whole network. Every node at level \( k \) maintains a tuple for itself in \( \beta_k \), and has at most \( n - k - 1 \) predecessors that do the same. This generates at most \( n - k \) tuples in the whole network. The highest level in the DAG network is \( D \), so the total required space is \( \sum_{i=0}^{D} (n - i) = (D + 1) \times n - \sum_{i=1}^{D} i = \frac{(D+1) \times (2n-D)}{2} \). Since \( D \leq n \) (the total number of nodes in the network), the average space complexity per node is \( O(D) \) tuples.

Theorem 3 (Stabilization Time) The system \( S_F \) stabilizes in \( O(D) \) time, where \( D \) is the diameter of the DAG network.

Proof: We assume that in each time unit, a node executes all rules that are enabled at that node. As shown in the proof of Lemma 3, a node may execute at most \( 2\delta + 1 \) rules within one time unit, where \( \delta \) is the out-degree of the DAG. Starting from an arbitrary configuration, in one time unit, no rule is enabled at any node in \( N_0 \). When all rules are disabled at all nodes up to level \( i \), the input to the nodes at level \( i \) cannot change. So, within one time unit, all rules at all nodes at level \( i + 1 \) are disabled. Continuing in this manner, after the highest level has been reached, no rules are enabled at any node in the DAG network and the system is stabilized. Since the maximum level of the DAG network is limited by its diameter \( D \), the system stabilizes in \( O(D) \) time.
5 Conclusions

For directed acyclic graph networks, we presented a self-stabilizing, parameterized protocol which is then instantiated with two functions to solve the topological sort and the shortest path routing protocols. More function parameters can be defined to solve other problems on DAG networks by this approach. One advantage of using our parameterized protocol is that only a partial (i.e., non-stabilizing) correctness analysis of the solution is necessary, while the solution is still self-stabilizing.

Let us now discuss the possibility of applying our algorithms to other topologies than DAG networks. As mentioned in Remark 1, we can construct a DAG system in an arbitrary connected undirected network, and then apply our methodology. We can also construct n minimum height spanning trees, rooted at every node in the general network. As such, our shortest path protocol can route messages only to the successor nodes in the DAG such constructed. However, it can be easily modified such that a node can send a message to its predecessor when none of its successors is able to route the message to the destination. Thus by running our shortest path algorithm on each of the spanning trees, each node will construct a shortest path routing table to route messages to any destination.

References